Biweely Report 5.1
CODE DOCUMENTATION
and UML diagrams

Raunak Narwal
Department of Mathematical Sciences
Indian Institute of Science Education and Research, Mohali, 130406, Punjab

December 12

Introduction

We have added comments, docstrings and created UML diagrams for all of our scripts. UML files were created using planttext website, it take PUML files and outputs PNGs and SVGs. I have added Numpy styled Docstrings using Pyment library, afterwards i have manually edited them to make them more descriptive and useful. I would try to include all the basics that build up to make our code. The information in the scripts, Readme file and this report overlaps. Each of them has their own purpose and is useful in its own way.

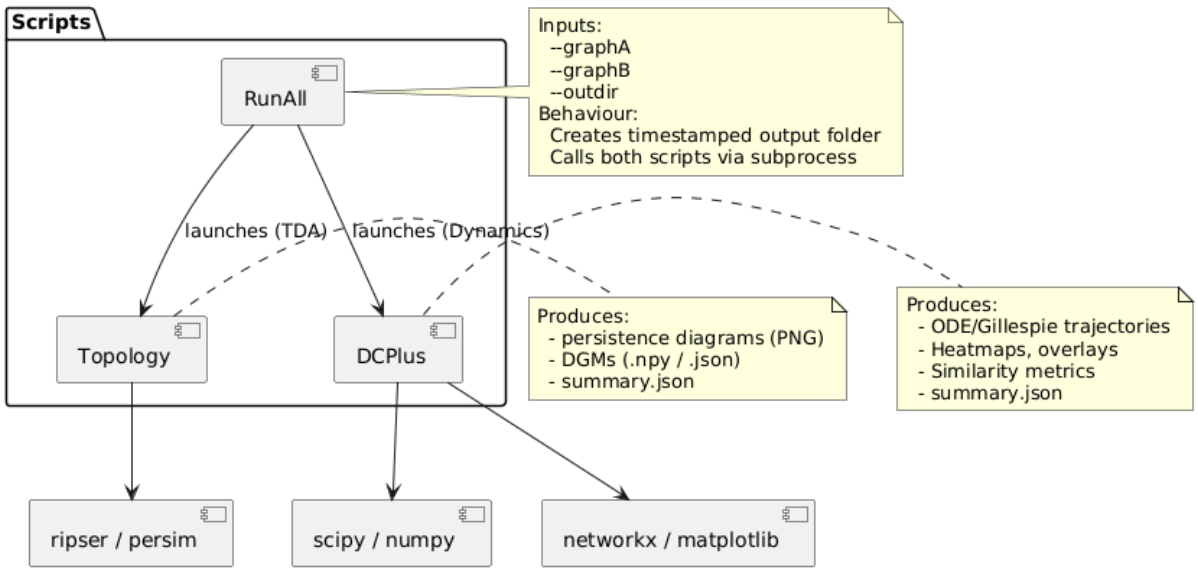

Figure 1: UML Diagram

Topology_Compare.py

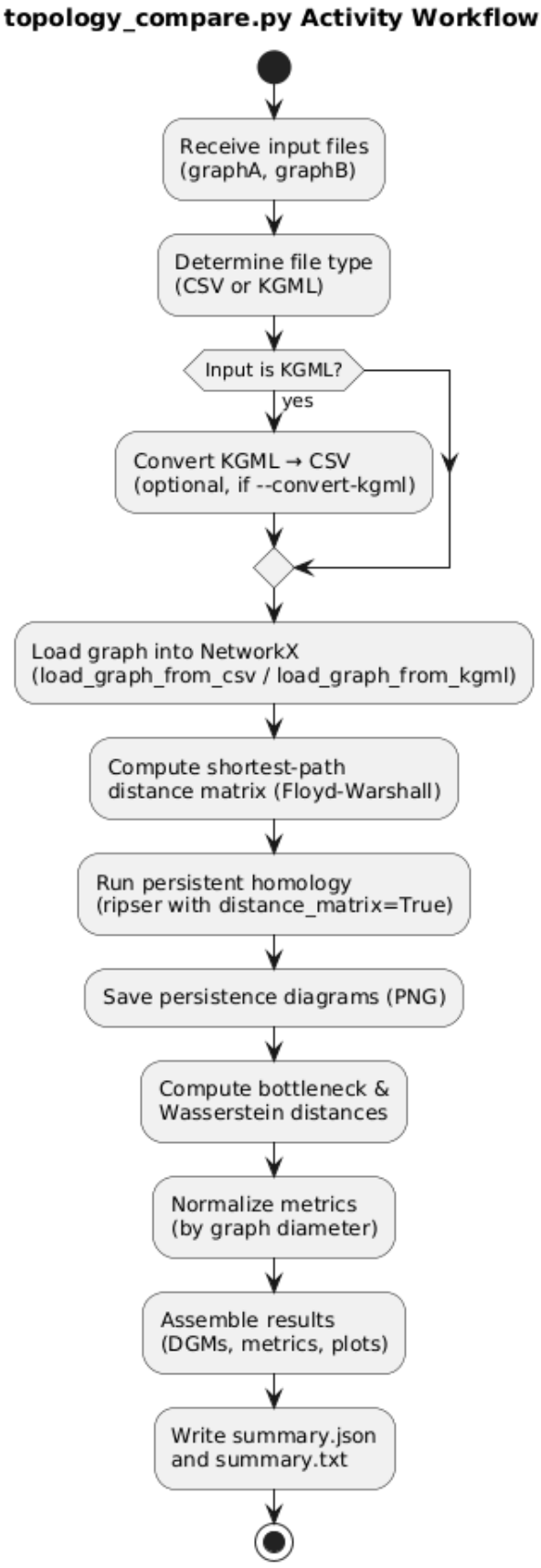

Figure 2: UML Diagram for Topology_Compare.py

Lines of Code Explained
The code imports argparse (for command line interface)
os/Pathlib for file and directory handling
json for saving metrics
numpy and pandas for data handling
networkx for graph operations
floyd warshall is useful in computing shortest path matrix
ripser is imported for persistent homology computation
persim for plotting and distances
seaborn for aesthethic graphs and plotting
xml.etree.ElementTree for parsing XML files

ensure_outdir(path)
This function checks if the output directory exists, if not it creates one.
It wraps a string path into a path object
calls mkdir method with exist_ok = True, it creates nested folders if needed and prevents errors if directory alreadt exists.

kgml_to_csv(kgml_path, out_csv)
This function converts pathways from KEGG KGML format into CSV format so that rest of the pipeline can easily read.
KGML contains entry notes : species , enzymes and compunds
relation edges : regulation links and reaction edges : metabolic conversions
The function works by parsing the XML tree structure of KGML file

```
tree = ET.parse(kgml_path)
root = tree.getroot()
```

It iterates over entries and relations to extract nodes and edges

```
entries[eid] = name
edges.append((source_name, target_name, rtype))
```

Extracts reactions

```
<reaction>
<substrate id="X"/>
<product id="Y"/>
</reaction>
```

the loop

```
for s in substrates:
    for p in products:
        edges.append((entries[s], entries[p], "reaction"))
```

Then it converts them into pandas DataFrames and saves them as CSV files.

load_graph_from_csv(csv_path)
This function loads a CSV file into NetworkX graph

```
df = pd.read_csv(csv_path)
df.columns = [c.lower() for c in df.columns]
```

it reads the CSV using pandas, creates a directed graph internally
Then it builds an undirected projection, because distance matrix must be symmetric
Persistent Homology for asymmetric graphs is not well defined.

load_graph_from_kgml(kgml_path)
this is similar to above but it directly constructs a networkX graph without saving CSV. the previous function outputs a CSV edge list but this outputs a graph, so they share logic but serve different parts of our pipeline.

graph_diameter(G)
This function computes the diameter of a graph G
which is the longest shortest path between any two nodes
it handles trivial cases

```
if G.number_of_nodes() == 0:
    return 0
```

if the graph is connected, then use NetworkX built in BFS diameter

```
if nx.is_connected(G):
    try:
      return int(nx.diameter(G))
```

and if it fails , then fallback

```
D = dict(nx.all_pairs_shortest_path_length(G))
maxd = max(d for u in D for d in D[u].values())
```

We compute diameter because later, bottleneck/wasserstein are normalized by graph diameter to make metrics comparable across networks.

shortest_path_distance_matrix(G, disconnected_value=None)
this is a very critical function in our pipeline, it converts the graph into N*N distance matrix which is then used by Ripser for computing persistent homology.
Lists nodes and indexing

```
nodes = list(G.nodes())
idx = {node: i for i,node in enumerate(nodes)}
```

this creates consistent ordering of nodes
initializes distance matrix with disconnected value (infinity by default)

```
D = np.full((n, n), np.inf)
np.fill_diagonal(D, 0)
```

fill matrix with edge weights

```
for u, v, data in G.edges(data=True):
    w = data.get("weight", 1.0)
    D[i,j] = w
    D[j,i] = w
```

floyd warshall algorithm is used to compute all pairs shortest paths
this computes all pairs shortest paths efficiently
replaces infinities with a large finite value

```
if disconnected_value is None:
    disconnected_value = 2 * max(finite_distances)
D[~np.isfinite(D)] = disconnected_value
```

persistence_from_distance(D, maxdim=1, thresh=None)
purpose is to run Ripser persistent homology on the distance matrix
it sets threshold for filtration.

```
thresh = 95th percentile of distances
```

calls ripser with distance matrix and parameters

```
res = ripser(D, maxdim=maxdim, thresh=thresh, distance_matrix=True)
```

it outputs the diagram in a list form

persistence_to_dict(dgms)
it converts diagrams into a dictionary format for easier storage and retrieval

```
{
"H0": [[birth, death], ...],
"H1": [[birth, death], ...]
}
```

plot_and_save(diagrams, title, outdir, tag)
creates a PNG image of the persistence diagram.
plot_graph(G, outdir, name)
this ceates the graphs visualisations
compute layout

```
pos = nx.spring_layout(G, seed=42)
cmap = plt.get_cmap('viridis')
```

degree based node size: nodes with higher degree appear larger
degree based color

compute_distance_metrics(dgA, dgB, p=2)
this computes the TDA distances between two diagrams
it computes bottleneck and wasserstein distances for each homology dimension

```
bn = bottleneck(A, B)
ws = wasserstein(A, B, p=p)
```

compare_diagrams_full(diagA, diagB, p=2)
it computes distances across all homology dimensions present

```
maxdim = max(len(diagA), len(diagB))
```

loops across dimensions

```
for dim in range(maxdim):
    A = diagA[dim]
    B = diagB[dim]
    bn, ws = compute_distance_metrics(A, B)
    metrics[f"H{dim}_bottleneck"] = bn
    metrics[f"H{dim}_wasserstein"] = ws
```

run_pipeline(graphA, graphB, nameA, nameB, outdir, maxdim, p)
this function integrates every stage, this is exatly what run_all.py calls.
ensures output directory

```
out = ensure_outdir(outdir)
```

Compute distance matrices

```
DA, nodesA = shortest_path_distance_matrix(graphA)
DB, nodesB = shortest_path_distance_matrix(graphB)
```

These produce: DA: NxN distance matrix for Graph A, DB: NxN distance matrix for Graph B and nodesA and nodesB: node lists in same order as matrices
then computes persistence diagrams

```
dgmA, resA = persistence_from_distance(DA, maxdim=maxdim)
dgmB, resB = persistence_from_distance(DB, maxdim=maxdim)
```

saves persistence diagrams

```
np.save(f"{nameA}_dgms.npy", dgmA)
np.save(f"{nameB}_dgms.npy", dgmB)
```

converts diagrams to JSON

```
json.dump(persistence_to_dict(dgmA), f)
json.dump(persistence_to_dict(dgmB), f)
```

```
{
"H0": [[0, 1], [0, 2], ...],
"H1": [[1.2, 3.5], ...]
}
```

saves all visual diagrams

```
figA = plot_and_save(dgmA, title=nameA, tag=f"{nameA}_tda")
figB = plot_and_save(dgmB, title=nameB, tag=f"{nameB}_tda")
```

compute raw topological distances
Normalize distances by graph diameter

```
diamA = graph_diameter(graphA)
diamB = graph_diameter(graphB)
diam_scale = max(diamA, diamB)
norm_metrics[k + "_norm"] = metrics[k] / diam_scale
```

Normalization prevents large networks from always having larger metric values

DC_plus.py

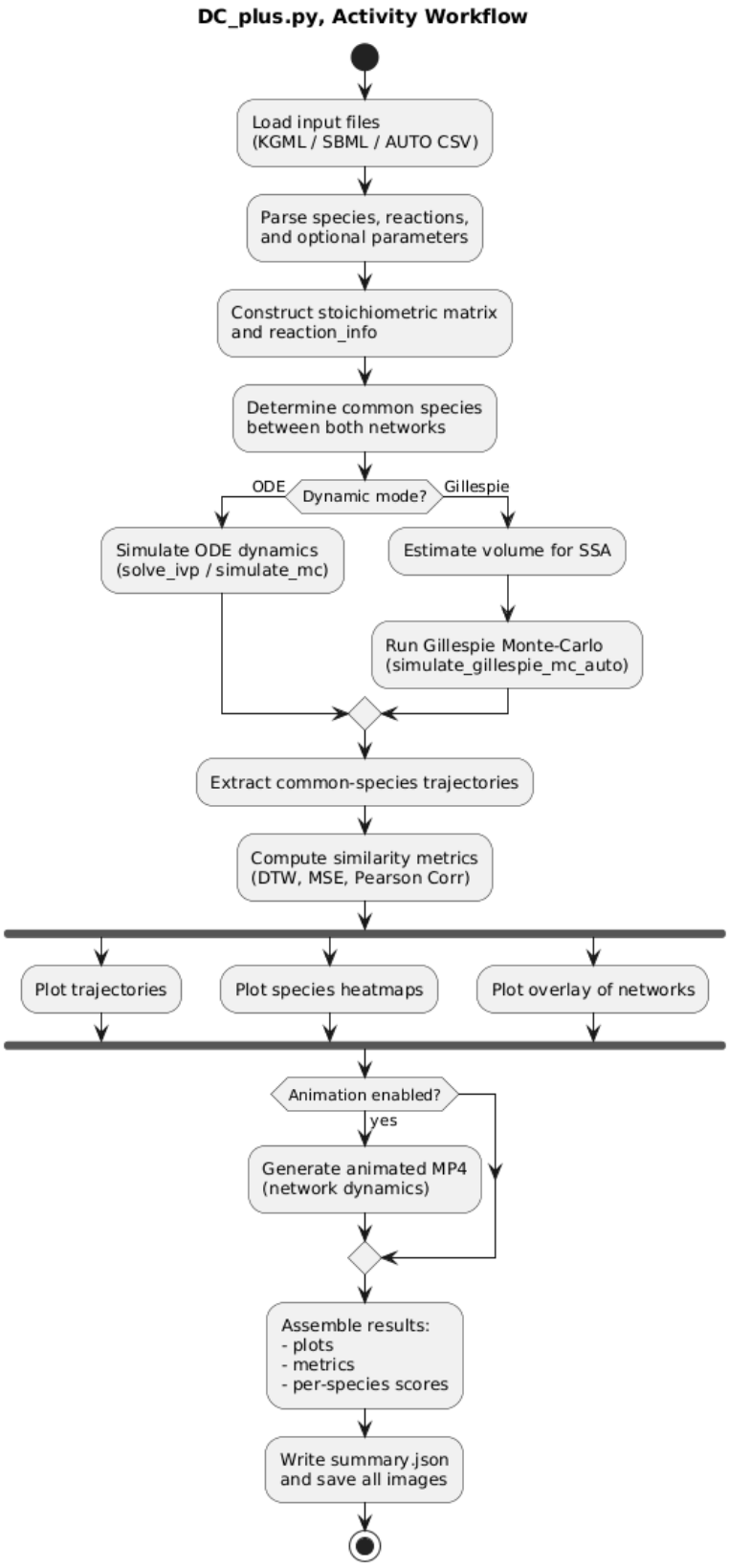

Figure 3: UML Diagram for DCPlus.py activity

Lines of Code Explained
Imports
xml.etree.ElementTree : read KGML/SBML XML
numpy, pandas : numerical ops
networkx : used for constructing reaction graphs
matplotlib, seaborn for plotting
argparse, json, os, sys for CLI + file handling
logging for debugging output
scipy.integrate.solve_ivp as an ODE solver
dtaidistance.dtw for dynamic time warping
scipy.stats.pearsonr for computing correlations
matplotlib.animation.FuncAnimation for animations

parse_kgml(kgml_file)
this function extract species and reactions from KGML files.
just like what Topology_Compare.py does
it parses the XML tree

```
tree = ET.parse(kgml_file)
root = tree.getroot()
```

Extract species

```
if entry.get('type') == 'compound':
   species_map[id] = name
```

each reaction is converted into a dictionary

```
{
    'id': rid,
    'substrates': [(sid, sto), ...],
    'products': [(pid, sto), ...]
}
```

it returns : species_list: list of KEGG IDs
reactions: list of substrate/product structures
species_map: mapping ID name

parse_sbml(sbml_file)
SBML files have different structure, very different in comparison to KGML
this functions extracts the same information from SBML format.
Identify SBML XML namespace

```
ns = {'sbml': 'http://www.sbml.org/sbml/level2/version4'}
```

Extract species

```
species_list = [s.get('id') for s in species_elems]
```

Extract reactions

```
<listOfReactants>
    <speciesReference species="A" stoichiometry="2"/>
</listOfReactants>
```

It returns the same 2 objects as KGML parser

parse_auto(file_path)
This function automatically detect if the file is KGML or SBML.
Detection works by reading the first 2KB of the file
if it contains "<sbml" tag, it is SBML
else if it contains "<pathway" tag, it is KGML
else it raises an error

load_params(params_path)
parameter loading is optional
purpose of this function is to load reaction rate constants (k) and model types
If JSON:

```
{
    "default_k": 1.0,
    "reactions": {
        "R1": {"k": 0.1, "type": "massaction"}
    }
}
```

If CSV:

```
reaction_id, k, type
R1, 0.2, massaction
R2, 0.05, massaction
```

Output format:

```
{
    'reactions': {
        'R1': {'k': 0.2, 'type': 'massaction'},
            ...
    },
    'default_k': <value or None>
}
```

Always returns a dictionary

build_ode_system(species_list, reactions, params=None, default_k=1.0)
Purpose of this function is to convert the parsed reaction list into a Stoichiometry matrix S, Reaction rate law definitions and a callable ODE function that computes dX/dt
This is the mathematical heart of our deterministic simulation.
Initialize

```
n = len(species_list)
m = len(reactions)
species_index = {sid: i for i, sid in enumerate(species_list)}
S = np.zeros((n, m))
```

Where:
n = number of species
m = number of reactions
species_index gives fast lookup
S = stoichiometric matrix
Fill stoichiometric matrix

```
S[idx, j] -= sto
S[idx, j] += sto
```

Load reaction parameters (k, type)

```
reaction_info = [
    {
        'id': 'R5',
        'k': 0.1,
        'type': 'massaction',
        'substrate_idxs': [0, 2],
        'product_idxs': [5],
        'sto_subs': [1, 2]
    },
        ...
]
```

Build the actual ODE function

```
def odes(t, x):
    v = np.zeros(m)
```

Mass-action kinetics

```
rate = k *  (x[species] ** stoich)
```

Rates: dX/dt

```
dxdt = S.dot(v)
return dxdt
```

Output is The ODE function, Species index mapping and Reaction metadata.

simulate(...)
Our ODE solver function
purpose is to integrate the ODE system over time using SciPy's solver_ivp
Default Behaviour: Use RK45 solver, Simulate between t_span = (0, 20), Evaluate n_points = 200 points and Start with initial concentration = ones vector
Adaptive "stop at steady state":

```
dx = ||x(t_n) - x(t_{n-1})||
if dx < threshold:
    stop early
```

If steady_threshold is provided:
Output is used for dynamic comparsion (DTW, MSE, correlations)

simulate_gillespie(...)
Monte Carlo simulation of reaction network using Gillespie SSA. This models discrete molecules counts with probabilistic behaviour.
Initialize molecule counts

```
X = 50 molecules for each species
```

Build update vectors

```
dx = vector showing molecule changes
e.g. A + B  C gives dx = [-1, -1, +1]
```

Compute propensities
$(a_j = k_j \prod (X_i)^{sto_i})$

Select next reaction

```
tau = -ln(r1)/a0
choose reaction j where cumulative_sum > r2 * a0
```

Apply update

```
X = X + updates[j]
t = t + tau
```

Output returns: times[ ] and states[ ][ ]

concs_to_counts and counts_to_concs
This function is created for better comparison, these convert between continuous ODE concentrations and integer SSA molecule counts.
conversion formula is: counts = concs * volume * Avogadro's number

convert_k_ode_to_k_ssa
ODE rate constants do not equal SSA rate constants
SSA works on counts, not concentrations.

estimate_volume_for_target_events(...)

automatically choose a simulation volume such that Gillespie SSA produces a reasonable number of reaction events.
too small volume gives too few events
too large volume gives too many events
this function estimates a volume such that the total reaction count is equal to target_events
Our method is to compute approximate rates using inital conditions

gillespie_setup_diagnostics(...)
Give debugging information for SSA: initial molecule counts, example propensities and first few reaction rates.

simulate_gillespie_mc_auto(...)
This function performs automatic volume estimation, gillespie Monte Carlo simulation, trajectory interpolation and production of mean standard deviation signals
Estimate Gillespie volume

```
V_required = estimate_volume_for_target_events(...)
```

Convert initial concentrations to counts
Convert ODE rates to SSA rates
Run multiple SSA simulations
Interpolate all runs onto a common fixed grid

```
np.interp
```

Output mean, std, and all trajectories

normalize_rows(y)

```
def normalize_rows(y):
    mx = np.max(y, axis=1, keepdims=True)
    mx[mx == 0] = 1.0
    return y / mx
```

it scales each species time series to [0, 1] range
because dynamic similarity measure (DTW, MSE, Correlation) are sensitive to absolute scales

compute_similarity(y1, y2)

```
def compute_similarity(y1, y2):
    n = min(y1.shape[0], y2.shape[0])
    y1n = normalize_rows(y1[:n])
    y2n = normalize_rows(y2[:n])
```

it only compares shared species count

```
n = min(#species in y1, #species in y2)
```

Normalize each species trajectory to [0,1] for DTW Correlation
Outputs dictionary, averages : dtw_avg, mse_avg, corr_avg
per species lists: dtw, mse and corr

save_plot(fig, path)

```
def save_plot(fig, path):
    fig.tight_layout()
    fig.savefig(path, dpi=300)
    plt.close(fig)
```

it is a centralised helper function to clean up figure layout, save consistently and close the figure to avoid memory leaks.

build_network_graph(...)

```
def build_network_graph(species_list, reaction_info, species_map=None):
    G = nx.DiGraph()
    for sid in species_list:
        G.add_node(sid)
```

purpose is to build a directed reaction graph, Nodes are species, edge from substrate are products
each edge stores the reaction ID
this is used by the animated network visualisation

animate_network_polished_v3(...)

```
def animate_network_polished_v3(G, species_list, reaction_info,
 t, y, pos, save_path,
    duration_sec=10, slow_factor=0.25):
    fig, ax = plt.subplots(figsize=(16, 12))
    palette = sns.color_palette("rocket", n_colors=len(species_list))
    node_color_map = {sid: palette[i] for i, sid in enumerate(species_list)}
    base_node_size = 400
    total_frames = len(t)
    fps = 30
    skip = max(1, total_frames // (duration_sec * fps))
    frame_indices = np.arange(0, total_frames, skip)
    interval = 1000 / fps / slow_factor
```

generates an mp4 animation of the network over time.
Each species (node) changes size depending on current concentration
nodes size is 400 + 5000* concentration
Based on instantaneous reaction rate derived from mass action formula.

plot_trajectories(...)

```
def plot_trajectories(t, y, species, title, save_path):
    fig = plt.figure(figsize=(12, 7))
    colors = sns.color_palette('tab20', max(4, len(species)))
    for i in range(y.shape[0]):
        plt.plot(t, y[i], label=species[i], lw=1.5,
        color=colors[i % len(colors)])
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', fontsize=8)
    plt.xlabel('Time')
    plt.ylabel('Concentration')
    plt.title(title)
    save_plot(fig, save_path)
```

Plots each species as a separate curve.
Uses tab20 palette, wraps colors if >20 species.
Moves legend outside to avoid clutter.
Saved via save_plot

plot_overlay(...)

```
def plot_overlay(t1, y1, t2, y2, species, save_path):
    fig = plt.figure(figsize=(14, 8))
    n = min(y1.shape[0], y2.shape[0])
    colors = sns.color_palette('tab20', max(4, n))
```

Shows Net1 vs Net2 for every common species
Solid curve is Net1 and Dashed curve is Net2

main()

```
parser = argparse.ArgumentParser(...)
parser.add_argument('file1')
parser.add_argument('file2')
...
args = parser.parse_args()
```

Reads all CLI parameters such as:
Which input files to compare
Whether to use ODE or Gillespie
How long to simulate
Whether to animate
Parameter file (k values)

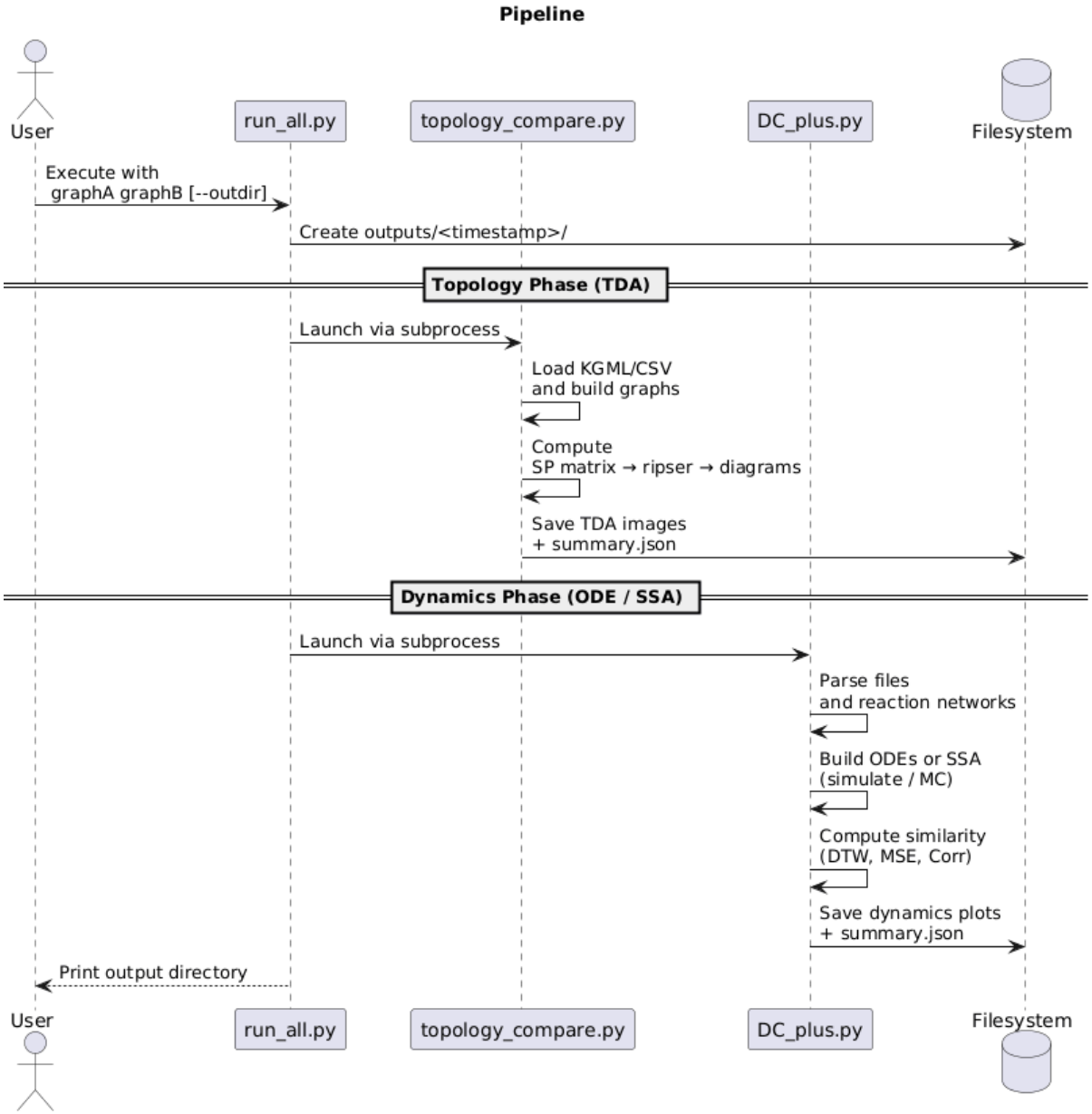

Figure 4: UML Diagram for Pipeline

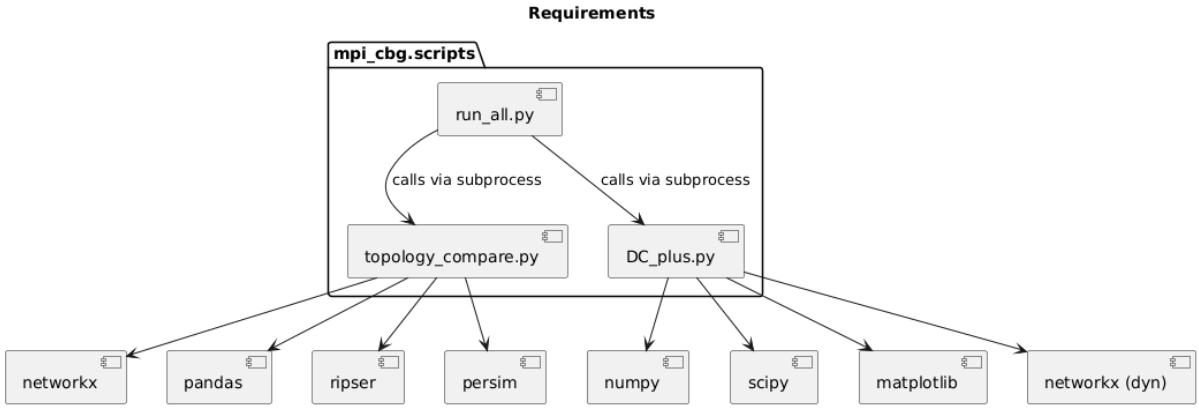

Figure 5

Generated: December 12, 2025